

Nº Orden	Apellido y nombre	L.U.	Cantidad de hojas
1	Brandwein Eric	349/16	6

A

MUY BUEN
PARCIAL

Organización del Computador 2

Segundo parcial - 14/11/17

CORRIGIO
POCHO

1 (20)	2 (50)	3 (30)	
18	30	35	83

Normas generales

- Numere las hojas entregadas. Complete en la primera hoja la cantidad total de hojas entregadas.
- Entregue esta hoja junto al examen, la misma no se incluye en la cantidad total de hojas entregadas.
- Está permitido tener los manuales y los apuntes con las listas de instrucciones en el examen. Está prohibido compartir manuales o apuntes entre alumnos durante el examen.
- Cada ejercicio debe realizarse en hojas separadas y numeradas. Debe identificarse cada hoja con nombre, apellido y LU.
- La devolución de los exámenes corregidos es personal. Los pedidos de revisión se realizarán por escrito, antes de retirar el examen corregido del aula.
- Los parciales tienen tres notas: I (Insuficiente): 0 a 59 pts, A- (Aprobado condicional): 60 a 64 pts y A (Aprobado): 65 a 100 pts. No se puede aprobar con A- ambos parciales. Los recuperatorios tienen dos notas: I: 0 a 64 pts y A: 65 a 100 pts.

Ej. 1. (20 puntos)

Responder detalladamente las siguientes preguntas, ejemplificar de ser posible.

- (4p) 1. ¿Cuántos bytes de tamaño, tiene un segmento de límite 0 y granularidad 0?
- (4p) 2. ¿Qué diferencia hay entre el bit *dirty* y el bit *accessed* en una entrada de tabla de páginas?
- (4p) 3. ¿Qué permisos efectivos tiene una página si su *Page Directory Entry* es de sólo lectura con nivel de usuario y su *Page Table Entry* es de lectura/escritura con nivel supervisor?
- (4p) 4. ¿Cuál es el nivel de privilegio necesario en un descriptor de interrupciones que atiende interrupciones de hardware? ¿Cómo se establece?
- (4p) 5. ¿Qué mecanismo se utiliza para saber si puedo acceder a un segmento de código?

Ej. 2. (30 puntos)

Se desea tener una función que dado una dirección de memoria correspondiente a la base del directorio de páginas, y una dirección física, devuelva un valor correspondiente a la cantidad de direcciones virtuales distintas por las que se puede acceder a dicha dirección.

La signatura debe ser

```
unsigned int cantidad_direcciones(unsigned int base_dir, unsigned int fisica);
```

- (15p) 1. Escriba el código en C de la función, contando solo direcciones que se puedan acceder en nivel Supervisor.
- (5p) 2. Modifique la función anterior para considerar las direcciones en nivel de Usuario.
- (10p) 3. Si el valor devuelto por los llamados a las funciones anteriores con una determinada dirección física son 0, ¿se puede asumir que la página que corresponde es una página física libre? Justifique.

Nota: Considerar que los marcos de página son de 4kb.

Ej. 3. (50 puntos)

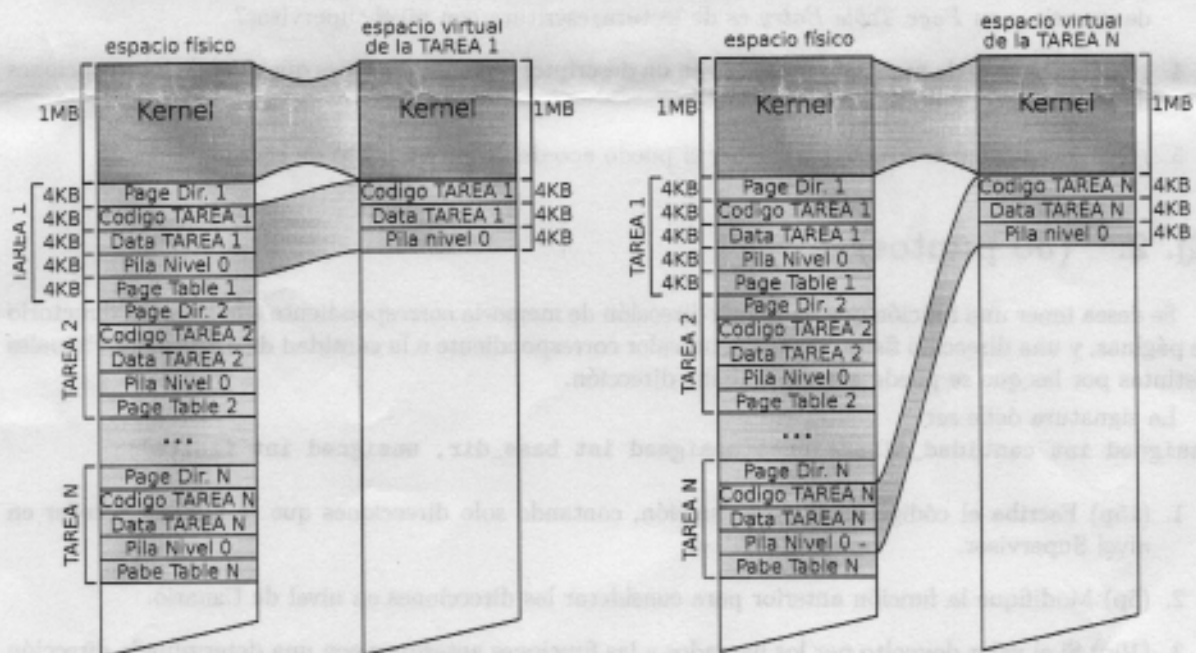
En un sistema con segmentación flat, se propone el esquema de paginación que muestra la figura a continuación. Cada tarea ocupa exactamente 20 KB de memoria física, que corresponden a 5 páginas.

El mapeo a memoria virtual de cada tarea corresponderá a mapear el código, datos y la pila de nivel 0 según corresponda a cada tarea.

Las tareas en el sistema son ejecutadas en orden, una por cada ciclo de reloj. Inicialmente las tareas comienzan con el EIP en la primer dirección de memoria de código y todos los registros de uso general en 0x00, excepto EAX, que contendrá el número de tarea que se esta ejecutando y EBX que contendrá el valor 0xFF la primera vez que se ejecute, o el código de error de la excepción correspondiente en el caso de que la tarea se haya reiniciado.

Las tareas pueden cometer cualquier tipo de excepción, en ese caso deben ser reiniciadas y comenzar a ejecutar inmediatamente respetando las condiciones descriptas en el párrafo anterior. Puede suponer que la pila de nivel 3 apunta la tope de la página de datos de nivel 3.

- (10p) 1. Indicar los campos relevantes de todas las estructuras involucradas en el sistema para administrar segmentación, paginación, tareas, interrupciones, privilegios, registros de control y funciones del scheduler. Explicar como deben instanciarse las estructuras de datos y explicar su funcionamiento.
- (15p) 2. Programar en C la función `mapear_tarea`, que dado el puntero al directorio de paginas de una tarea se encarga de construir todo el mapa de paginación de la misma.
- (15p) 3. Programar en ASM/C la rutina de atención de interrupciones de alguna excepción del procesador.
- (10p) 4. Programar en ASM/C la rutina de atención de interrupciones del reloj.



1. Tiene 1 solo byte de tamaño. ✓
2. El bit dirty se setea en 1 cuando algún proceso escribe en una dirección perteneciente a la página indicada por esa entrada, y el bit accessed se setea cuando alguna de esas páginas es leída ^{o ejecutada} o escrita. Ambos pueden ser seteados en 0 por el sistema operativo si se desea; el procesador no lo hace.
3. Si CR0.WP está en 1, tanto para el kernel como para el usuario los permisos efectivos serán los más restrictivos, es decir, de sólo lectura con nivel supervisor. Si CR0.WP estuviera en 0, el kernel podría escribir la página aunque sea de sólo lectura.
5. Se utilizan el mecanismo de chequeo ^{nivel de} protección, asegurando que el bit de protección está activado en el CR0. Cuando un proceso realiza una instrucción que implica correr código de la manera jmp far o call far, la dirección lógica proveída por el mismo contiene un selector de segmento y un offset. El selector de segmento, a su vez, contiene el RPL en sus primeros dos bits, que es el requested privilege level. Si asumimos que el segmento de código ~~no es conforming~~ obtenido con el índice del selector de segmento no es conforming, el chequeo se realiza con estos pasos:

$$NO \quad CPL == DPL$$

 - Se calcula el $EPL = \max(CPL, RPL)$, CPL siendo el descriptor privilege level del segmento de código del proceso que realizó el pedido; sigue

• Se compara el EPL obtenido con el DPL del segmento de código al que se está midiendo acceso, permitiéndolo si y sólo si $EPL = DPL$. Si, en cambio, asumimos que el segmento es conforming, el cálculo cambia. En vez de calcular el EPL, se usa directamente el CPL del proceso que hace el pedido, y se compara con el DPL del segmento de código pedido. Si $CPL \geq DPL$, el código pedido pasa a ejecutarse con el nivel de privilegio del proceso original.

4- El nivel del DPL de este tipo de descriptores debe ser 0.

El programador del kernel, cuando crea cada uno de los entres de la IDT, en las posiciones de las interrupciones externas, debe setear el DPL en 0, que son los bits 21 y 22.

```

1.
unsigned int cantidad_direcciones(unsigned int base_dir,
    unsigned int fisica) {
    // asumo que base_dir es la dirección física de la base,
    // y que tiene los 12 bits menos significativos en 0.
    // para no tener que hacer la función 2 veces por el punto
    // 1. y 2., delego a otra función que tome el privilegio
    // como parámetro
    return cantidad_direcciones_con_priv(base_dir, fisica, 0);
}

unsigned int cantidad_direcciones_con_priv(unsigned int base_dir,
    unsigned int fisica, unsigned int u_s) {
    pte* directorio = (pte*) base_dir;
    unsigned int cantidad = 0;
    for (unsigned int i = 0; i < 1024; i++) {
        if (directorio[i].p == 1) {
            pte* tabla = (pte*) (directorio[i].addr << 12);
            for (unsigned int j = 0; j < 1024; j++) {
                if (tabla[j].p == 1 &&
                    fisica_accessible(directorio[i], tabla[j],
                        fisica, u_s)) {
                    cantidad++;
                }
            }
        }
    }
    return cantidad;
}

bool fisica_accessible(pte dir_entry, pte table_entry,
    unsigned int fisica, unsigned int u_s) {
    bool accesible = u_s <= dir_entry.user_supervisor &&
        u_s <= table_entry.user_supervisor;
    unsigned int base_pagina = table_entry.addr << 12;
    unsigned int ultima_dir = base_pagina + 0xFFF;
    return accesible && base_pagina <= fisica &&
        fisica <= ultima_dir;
}

```

parvistas!

Se asume que los structs Ade y pte son proveídos, y la cantidad de vuelta se requiere a todas las direcciones que se pueden acceder con modo supervisor.

2. Gracias a que delegamos a otra función, única que hay que cambiar para ~~que~~ contar las direcciones que solo se pueden acceder en modo usuario es llamar a cantidad_direcciones_con_priv con el último parámetro igual a 1. ✓

3. Excepto que la base_dir apunte al único directorio de tablas que hay en todo el sistema, no se puede asumir que esté libre. ~~Es más, aunque se le dé el único, este no es~~ y aunque fuese el único, es posible que el kernel, en su gran sabiduría, use una dirección ^{física} marcando cada vez una nueva dirección virtual a la ^{física} que quiere utilizar, y luego de utilizarla la desmarque, por ejemplo. ✓

Si el directorio no fuese el único, podría haber otros directorios que tengan marcada la ^{página} ~~dirección~~, con lo cual no sería una página libre. ✓

1. Segmentación

Debe haber una GDT ~~en~~ con 5 entradas iniciales: Un descriptor nulo, y los otros 4 de esta manera:

	G	D/B	P	DPL	S	Type	Limit
(Data W. 0)	1	1	1	0 0	1	0010	0xFFFF
(Código W. 0)	1	1	1	0 0	1	1000	0xFFFF
(Data W. 3)	1	1	1	1 1	1	0010	0xFFFF
(Código W. 3)	1	1	1	1 1	1	1000	0xFFFF

Para el limit, se asume que la memoria tiene 4 GB

Estos cuatro descriptors tienen base = 0.

Para cada una de las N tareas deberemos tener una entrada de descriptor de TSS, y otra para la tarea inicial, pero eso se discutirá en la sección de tareas.

La dirección física base de la GDT deberá ser cargada en el registro GDTR, en CS se deberá cargar 0x10, y en los otros registros de selectores de segmento, 0x0.

Paginación

Se utilizará paginación de dos niveles, teniendo cada tarea su Page Directory y ^{una} Page Table. El Page Directory tendrá una sola entrada con $P=1$, $R/W=1$, $U/S=1$, y $addr$ = los 20 bits más significativos de la dirección física ^{base de la} de la Page Table de la tarea en particular, y los demás bits = 0. En la Page Table, los primeros 256 entradas serán mapeados con identity mapping a las direcciones físicas del kernel, con lo cual tendrán $P=1$, $R/W=1$, $U/S=0$, $addr$ = los 20 bits más significativos de la base de la página correspondiente, y el resto de los bits en 0. Las siguientes tres entradas apuntarán al código, datos, y la pila de nivel 0 de cada tarea, y entonces serán así:

sigue

empezando desde 0

Entrada #	P	R/W	U/S	Addr.
Código: 256	1	0	1	$256 + N*5 - 5 + 1$
Datos: 257	1	1	1	$256 + N*5 - 5 + 2$
Pila: 258	1	1	0	$256 + N*5 - 5 + 3$

N = número de tarea actual.

Para todos los otros entresados, tanto del PD como de la PT, tendremos $P=0$.

Para habilitar paginación, necesitaremos también tener un PD y ~~una~~ Page Tables del kernel. Por simplicidad, construiremos estas estructuras con identity mapping de todo el espacio de memoria del kernel. También necesitaremos setear el CR3 con la base del PD del kernel, y setear los bits $CR0.PE=1$ y $CR0.PG=1$, para habilitar modo protegido y paginación.

TAREAS

Por cada una de los N tareas, como dicho anteriormente, deberemos tener una entrada en la GDT que señale los descriptores de TSS, y otra entrada más para la tarea inicial.

El descriptor de TSS de la tarea inicial sólo deberá apuntar, con su base, a un espacio en memoria con ~~el~~ espacio suficiente libre consecutivo para poder guardar una TSS, ya que esto será "nuestro" cuando se quite a otra tarea. Deberá, también, tener $P=1$. Los descriptores de TSS, de las tareas, en cambio, deberán apuntar a ~~las~~ direcciones con TSSs cargados con datos específicos; en particular, cada TSS deberá tener:

- $ioomap = 0xFFFF$
- $EFLAGS = 0x202$, para habilitar interrupciones
- $GS, FS, DS, SS, ES = 0x1B$
- $CS = 0x23$
- $EIP = (256 + n*5 - 5 + 1) \ll 12$, siendo n el número de tarea ~~al que pertenece la TSS~~
- $EAX = n$
- $EBX = 0xFF$
- $EDI, ESI, EDI, ECX = 0$
- $EBP = (256 + n*5 - 5 + 3) \ll 12$
- $SS0 = 0x0$
- $ESP0 = 256 + n*5 - 5 + 4$

Hoja 4/6

El resto de la TSS estará en 0.

Para comenzar a correr las tareas, se deberá cargar el Task Register con el selector de segmento de ~~la~~ la TSS de la tarea inicial, sea, como el descriptor está en ~~el~~ el índice 3 de la GDT, es 0x28. Luego, se habilitarán interrupciones como se verá en la sección de interrupciones, y se hará un jmp far a la primera tarea, con la dirección lógica corresnadiante.

Interrupciones

Se verá haber una estructura, llamada IDT, que tendrá como entradas descriptors de interrupciones. Los índices 0 a 31 de esta estructura serán interrupciones de excepción del tipo 0xE, sea, 32-bit interrupt gates, que apuntarán al código a ejecutar cuando sucede una ~~int~~ excepción. El índice 32 tendrá otro descriptor de tipo 32-bit interrupt gate, que atenderá a interrupciones de reloj, y apuntará el código correspondiente. Todos tendrán P=1 y DPL=00.

Para habilitar interrupciones, se deberá cargar con lidt el registro IDTR, que contendrá la base y el límite de la IDT. Luego, se deberá llamar a sti, ~~se habilita~~.

privilegios

Se generaron los privilegios entre el kernel, ~~de~~ de nivel 0, y el usuario, de nivel 3. Los niveles de cada estructura ya fueron asignados según correspondía en los otros secciones.

Registros de Control

Ya se explicó el CR3, y así todos los registros pertenecientes del CR0, excepto el WP, que será igual a 1. Los otros registros de control también fueron explicados.

Funciones del Scheduler

El scheduler actuará en cada interrupción de reloj, calculando la siguiente tarea y pasando a ejecutar esa, y en cada excepción, reiniciando la tarea interruptora, cargando ~~en el registro EBX~~

EBX = código de error.

2.

void mapear_tarea(pte* dir, unsigned int tarea) {

dir[0].p = 1;

dir[0].real_write = 1;

dir[0].user-supervisor = 1;

dir[0].addr = dir_paginas_tarea(tarea) + 4;

dir[0].access = 0;

dir[0].ignore = 0;

dir[0].page-size = 0;

dir[0].ignore2 = 0;

dir[0].cache-disable = 0;

dir[0].write-through = 0;

dir[0].global = 0;

for (unsigned int i = 1; i < 1024; i++) {

dir[i].p = 0;

}

pte* tabla = (pte*) (dir[0].addr << 12);

inicializar_tabla(tabla);

mapear_kernel(tabla);

mapear_paginas_tarea(tabla, tarea);

}

unsigned int dir_paginas_tarea(unsigned int tarea) {

return 256 + tarea * 5 - 5;

}

void mapear_kernel(pte* tabla) {

for (unsigned int i = 0; i < 256; i++) {

tabla[i].p = 1;

tabla[i].real_write = 1;

tabla[i].user-supervisor = 0;

tabla[i].addr = i << 12;

}

}

```

void inicializar_tabla (pte* tabla) {
    for (unsigned int i = 0; i < 1024; i++) {
        tabla[i].p = 0;
        tabla[i].accessed = 0;
        tabla[i].dirty = 0;
        tabla[i].cache_disable = 0;
        tabla[i].write_through = 0;
        tabla[i].pat = 0;
        tabla[i].global = 0;
        tabla[i].ignores = 0;
    }
}

```

```

void mepeor_paginas_tarea (pte* tabla, unsigned int tarea) {
    tabla[256].p = 1;
    tabla[256].read_write = 0;
    tabla[256].user_supervisor = 1;
    tabla[256].addr = dir_paginas_tarea(tarea) + 1;

    tabla[257].p = 1;
    tabla[257].read_write = 1;
    tabla[257].user_supervisor = 1;
    tabla[257].addr = dir_paginas_tarea(tarea) + 2;

    tabla[258].p = 1;
    tabla[258].read_write = 1;
    tabla[258].user_supervisor = 0;
    tabla[258].addr = dir_paginas_tarea(tarea) + 3;
}

```

Se asume que los struct pte y pte están dados.

3. ASM:

section .data

selector: dw 0x0000

offset: dw 0x00000000

section .text

call revidor_difer

int ecx

push esp, ignore error code

mov ecx, 0x10000000

mov [esp+0], ecx

call esp_tarea, 4 + 40

mov [esp+0], ecx

pop esp

ret 4, ignore error code

iret

07/04/14

C: 09:00

No me alcanzó el tiempo.

4- ASM:

section .data

tss_offset: dd 0x00000000

tss_selector: dw 0x0000

section .text

pushad

call pic_fin_int

call sched_tarea_siguiente

mov [tss_selector], ax

jmp far [tss_offset]

popad

iret

C:

uint sched_tarea_siguiente() {

tarea_actual = (tarea_actual + 1) % N;

return (0x5 + tarea_actual) << 3 + 3;

}

Donde tarea_actual es un unsigned int con la tarea actual, y N es la cantidad de tareas, se asume que hay más de una tarea, y entonces nunca se llama a la misma tarea que está corriendo.